University of Edinburgh, School of Informatics

Secure Programming Coursework: Part 1

---

2. Secure Coding

Question 1

- *CWE-121:* Stack-based Buffer Overflow
  Description: Line 25, strcat(greeting, user); The `user` length (50) exceed the capacity of `greeting` (30)
- *CWE-134:* Use of Externally-Controlled Format String
  Description: Line 29, printf(greeting); The variable `greeting` might contain both output and string format notations.

We can identify two vulnerabilities, but the harm mainly occurs from the second (line 29). The attackers might abuse the fact that the formatting instructions are not given and input their own. They can use the `%n` notation to write to an address they choose, passing it through the input.

Question 3

We first used the command `objdump -t ./vulnerable` to find the address of variable `board`. After that, we add five to it to get the address of the digit we want to change (we want to change "1" to "2").

Our goal is to exploit the `printf(greeting)` command in line 27 of `vulnerable.c`. If we look at the documentation of `printf`, we can see that it receives first the formating parameters and then the address of a variable to display. However, we can take advantage of the `%n` notation. `printf` will accept the count of any characters before that notation and write it to the indicated address.

The main idea behind the exploit was to pass to the program the address of `board` and align it in the way described above so that we can change `board1` to `board2`. After testing the `vulnerable` program, by inputing the `board` address followed by several `%x` notations, we found that the address is stored on the ninth place over the stack. So given that we have

already input the hardcoded three characters as word alignment, the precalculated address of `board[5]`, the `%9$n` notation, and another six characters that are already included in the greeting message ("Hello "), we need to suffice another 36 to reach 50. 50 is translated to `2` in ASCII, which is the value we want to write on the address where the 5th character of the board buffer is.

The exploit script should output a vulnerable input that looks like this `<space><space><space>\xb5\x99\04\x08%36x%9$n`. The message input can have any value.

Question 5

Vulnerability 1:

- **Problem:** In `vulnerable2.c` we can see a call to the function `gets()` at lines 34 and 35.
- **Description:** Using `gets()` is dangerous since it can lead to buffer overflow attacks. In more detail, `gets()` takes as input a pointer to a buffer where it stores any user input.
- **Exploitation:** If the input data is much larger than the buffer size, we can overflow it and cause memory corruption.
- **Correction:** Instead of `gets()`, we should use `fgets()`, which is much safer since we can limit the number of characters that we can read as input.

Vulnerability 2:

- **Problem:** In `vulnerable2.c` we can see a call to the function `fprintf()` at line 28.
- **Description:** Using `fprintf()` with only two arguments is dangerous since it can leak data from the memory to the output file. In more detail, `fprintf()` takes as input a file pointer, format settings, and the data to write. In our case, only two arguments are given in line 28, which can be exploited.
- **Exploitation:** If the message data contains only formatting characters such as `%x`, the `fprintf()` function will use them as the second argument and fetch actual data from the stack to print in the output file.
- **Correction:** We can replace `fprintf(file, mess)` with `fprintf(file, "%s", mess)`

Vulnerability 3:

- **Problem:** In `vulnerable2.c` we can see that both user and password are defined on lines 6 and 7.
- **Description/Exploitation:** An adversary using the `objdump -t ./vulnerable2` command can extract any hardcoded strings defined in the program. Hence, determining the user and password like that is not safe. This can lead to unauthorized access.
- **Correction:** Instead of defining the plain user and password texts, we can encode them using a hash function, like SHA1. The adversary will see the hash digest but won't extract the password out of it. When a user inputs its credentials, we calculate the input hash and compare it to the already defined one.